Introduction to Slurm

Dr. Chennakesava Kadapa

Zienkiewicz center for Computational Engineering
Swansea University, Swansea, UK.

Email: c.kadapa@swansea.ac.uk

Outline

Introduction

Slurm is a fictional soft drink in the **Futurama** multiverse. It is popular and highly addictive. It is Philip J. Fry I's favourite drink.

Introduction

- Originally stood for **S**imple **L**inux **U**tility for **R**esource **M**anagement; not used any longer.
- Slurm is an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for Linux clusters.
- Slurm requires no kernel modifications for its operation and is relatively self-contained.
- Slurm is used on many of the world's top 500 supercomputers.
- As a cluster workload manager, Slurm has three key functions.
  1.) It allocates exclusive and/or non-exclusive access to resources (compute nodes) to users for some duration of time so they can perform work.
  2.) It provides a framework for starting, executing, and monitoring work (normally a parallel job) on the set of allocated nodes.
  3.) It arbitrates contention for resources by managing a queue of pending work.
- Main source for downloads and documentation: `https://schedmd.com/`
- Good material at `https://hpc.llnl.gov/training/tutorials#training_materials`

Architecture

Slurm consists of a **slurmd** daemon running on each compute node and a central **slurmctld** daemon running on a management node (with optional fail-over twin).

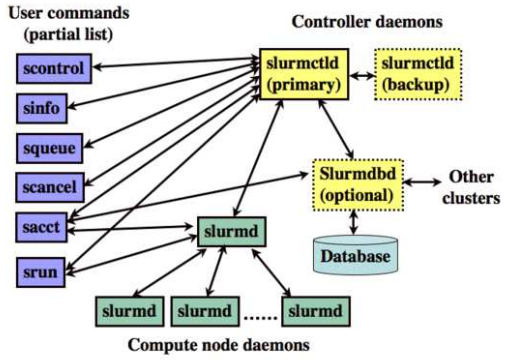

Figure 1. Slurm components

Important user commands include:

command	description
sacct	report job accounting information about active or completed jobs
salloc	allocate resources for a job in real time
sbatch	submit a job script for later execution (the script typically contains one or more srun commands to launch parallel tasks)
scancel	cancel a pending or running job
sinfo	reports the state of partitions and nodes managed by Slurm (it has a variety of filtering, sorting, and formatting options)
squeue	reports the state of jobs (it has a variety of filtering, sorting, and formatting options), by default, reports the running jobs in priority order followed by the pending jobs in priority order
srun	used to submit a job for execution in real time

Architecture

Slurm entities:

- **nodes:** the compute resource in Slurm,
- **partitions:** logical (possibly overlapping) sets of nodes,
- **jobs:** allocations of resources assigned to a user for a specified amount of time, and
- **job steps:** sets of (possibly parallel) tasks within a job.

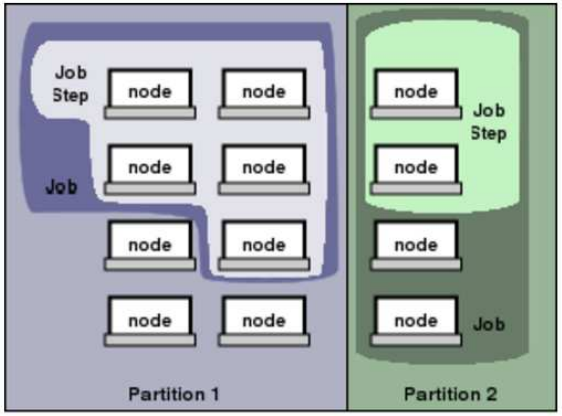

Figure 2. Slurm entities

The partitions can be considered job queues, each of which has an assortment of constraints such as job size limit, job time limit, users permitted to use it, etc. Priority-ordered jobs are allocated nodes within a partition until the resources (nodes, processors, memory, etc.) within that partition are exhausted. Once a job is assigned a set of nodes, the user is able to initiate parallel work in the form of job steps in any configuration within the allocation. For instance, a single job step may be started that utilizes all nodes allocated to the job, or several job steps may independently use a portion of the allocation.

Submitting jobs

Jobs typically specify what resources are needed, such as type of machine, number of machines, job duration, amount of memory required, account to charge, etc.

- The jobs can be submitted directly from the command prompt but this is quite cumbersome and difficult to maintain. So, the preferred method is to submit jobs using a script file.
- The jobs are submitted to the execution queue with the commands **sbatch** <**script_file**>. The queueing system prints a number (the job id) almost immediately and returns control to the linux prompt. At this point the job is in the submission queue.
- Once the job is submitted, it will sit in a pending state until the resources have been allocated to the job. The progress of the job can be monitored using the **squeue** command.

Script file

- A job script is a plain text file that you create with your favorite editor.
- Job scripts can include any/all of the following:
  - Commands, directives and syntax specific to a given batch system
  - Specific job details - directories, job name, output file name etc.
  - Requests for computer resources for the job - number of CPUs, amount of memory, etc.
  - Shell scripting - variables, loops etc.
  - Environment variables.
  - Names of executable(s) and input arguments, if any.
  - Comment lines and white space.
- Slurm options are specified with `#SBATCH`
- All `#SBATCH` lines must come before shell script commands.
- `#SBATCH` token is case sensitive. Using anything else will result in getting default settings.

Script file - a sample

The script starts with `#!/bin/bash`, usually called a **shebang**, which makes the submission script a Linux bash script.

```
#!/bin/bash
# set name of the job
#SBATCH --job-name=mytest
# set the number of nodes
#SBATCH --nodes=4
# number of processes per node
#SBATCH --ntasks-per-node=16
# set max wallclock time (D-hh:mm:ss)
#SBATCH --time=00:40:00
# load the required modules
module load cgal/4.1
# set the PATH variables
export LD_LIBRARY_PATH=...
# run the application
echo My application started
myCode
echo My application finished
```

Script file - #SBATCH options

- job name

```
#SBATCH --job-name="name"
```

- number of nodes

```
#SBATCH --nodes=2
```

- number of processes per node

```
#SBATCH --ntasks-per-node=16
```

- maximum walltime

```
#SBATCH --time=hh:mm:ss
```

  A job is expected to finish before the specified maximum walltime. Once the walltime reaches the specified maximum, the job terminates regardless of whether the job processes are still running or not.

- memory (in MB)

```
#SBATCH --mem=1024
```

Script file - #SBATCH options

- mail alert at start, end and abortion of execution

```
#SBATCH --mail-type=ALL
```

- mail to this address

```
#SBATCH --mail-user=<emailID>
```

Monitoring jobs

- **squeue** is the main command for monitoring the state of systems, groups of jobs or individual jobs. **scontrol** also provides some features for monitoring jobs. The command **squeue** prints the list of current jobs. The list looks something like this:

JOBID	PARTITION	NAME	USER	ST	TIMES	NODES	NODELIST

List all current jobs for a user	squeue -u <username>
List all running jobs for a user	squeue -u <username>-t RUNNING
List all pending jobs for a user	squeue -u <username>-t PENDING
List all current jobs in the general partition for a user	squeue -u <username>-p general
List detailed information for a job (useful for troubleshooting)	scontrol show jobid -dd <jobid>

Controlling jobs

scancel and **scontrol** are the main commands for controlling the jobs.

To cancel one job	scancel <jobid>
To cancel all the jobs for a user	scancel -u <username>
To cancel all the pending jobs for a user	scancel -t PENDING -u <username>
To cancel one or more jobs by name	scancel –name myJobName
To pause a particular job	scontrol hold <jobid>
To resume a particular job	scontrol resume <jobid>
To requeue (cancel and rerun) a particular job	scontrol requeue <jobid>

Environment variables

At the time a job is launched into execution, **Slurm** defines multiple environment variables, which can be used from within the submission script to define the correct work-flow of the job. The most useful of these environment variables are the following:

- **SLURM_SUBMIT_DIR**, which points to the directory where the sbatch command is issued. SLURM_SUBMIT_DIR can be useful in a submission script when files must be copied to/from a specific directory that is different from the directory where the **Slurm** command was issued.
- **SLURM_JOB_NODELIST**, which returns the list of nodes allocated to the job.
- **SLURM_JOB_ID**, which is a unique number **Slurm** assigns to a job. SLURM_JOB_ID is useful to tag job specific files and directories, typically output files or run directories.

MPI_Bcast

```
MPI_Bcast(
    void*           message    /* in/out */
    int             count      /* in     */
    MPI_Datatype    datatype   /* in     */
    int             root       /* in     */
    MPI_Comm        comm       /* in     */)
```

```
int main()
{
  // Define variables at the beginning
  // of the block, as in C:
  CStash intStash, stringStash;
  int i;
  char* cp;
  ifstream in;
  string line;
[...]
```